

**RAJALAKSHMI ENGINEERING COLLEGE**  
**AN AUTONOMOUS INSTITUTION**  
**Affiliated to ANNA UNIVERSITY**  
**Rajalakshmi Nagar, Thandalam,**  
**Chennai-602105**



**RAJALAKSHMI**  
**ENGINEERING COLLEGE**  
An AUTONOMOUS Institution  
Affiliated to ANNA UNIVERSITY, Chennai

**DEPARTMENT OF COMPUTER SCIENCE**  
**AND ENGINEERING**

**CS19641 COMPILER DESIGN LABORATORY**  
**ACADEMIC YEAR: 2023-2024 (Even)**

**NAME: GOKULA KRISHNA H**

**REGISTER NUMBER: 2116210701062**

**INDEX**

| <b>Exp<br/>No</b> | <b>Name of the Experiment</b>  | <b>REMARKS</b> |
|-------------------|--|----------------|
| 1                 | Develop a lexical analyzer to recognize tokens in C.<br>(Ex. identifiers, constants, operators, keywords etc.) |                |
| 2                 | Design a Desk Calculator using LEX.  |                |
| 3                 | Recognize an arithmetic expression using LEX and YACC.   |                |
| 4                 | Evaluate expression that takes digits, *, + using YACC   |                |
| 5                 | Generate Three address codes for a given expression<br>(arithmetic expression, flow of control).               |                |
| 6.a               | Implement Code Optimization Techniques like dead code<br>elimination, Common sub expression elimination        |                |
| 6.b               | Implement Code Optimization Techniques like copy<br>propagation  |                |
| 7                 | Generate Target Code (Assembly language) for the given set<br>of Three Address Code.                           |                |

## EXP 1 - DEVELOP A LEXICAL ANALYZER TO RECOGNIZE TOKENS IN C

### AIM:

To implement the program to identify C keywords, identifiers, operators, end statements like ; , [ ], { } using C tool.

### ALGORITHM:

- We identify the basic tokens in c such as keywords, numbers, variables, etc.
- Declare the required header files.
- Get the input from the user as a string and it is passed to a function for processing.
- The functions are written separately for each token and the result is returned in the form of bool either true or false to the main computation function.
- Functions are issymbol() for checking basic symbols such as () etc , isoperator() to check for operators like +, -, \*, / , isidentifier() to check for variables like a,b, iskeyword() to check the 32 keywords like while etc., isInteger() to check for numbers in combinations of 0-9, isnumber() to check for digits and substring().
- Declare a function detecttokens() that is used for string manipulation and iteration then the result is returned from the functions to the main. If it's an invalid identifier error must be printed.
- Declare main function get the input from the user and pass to detecttokens() function.

### PROGRAM:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#define MAX_CODE_LENGTH 1000
#define MAX_TOKEN_LENGTH 100

typedef enum {
    KEYWORD,
    IDENTIFIER,
    CONSTANT,
    OPERATOR,
    PUNCTUATION
} TokenType;

const char *keywords[] = {"int", "float", "void", "if", "else", "while", "for"};
const char *operators = "+-*/=<>";
const char *punctuation = ";(),{}";

int is_keyword(const char *token) {
    for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); ++i) {
        if (strcmp(token, keywords[i]) == 0) {
            return 1;
        }
    }
    return 0;
}
```

```

int is_operator(char token) {
    return strchr(operators, token) != NULL;
}

int is_punctuation(char token) {
    return strchr(punctuation, token) != NULL;
}

TokenType get_token_type(const char *token) {
    if (is_keyword(token)) {
        return KEYWORD;
    } else if (isdigit(token[0])) {
        return CONSTANT;
    } else if (isalpha(token[0]) || token[0] == '_') {
        return IDENTIFIER;
    } else if (is_operator(token[0])) {
        return OPERATOR;
    } else if (is_punctuation(token[0])) {
        return PUNCTUATION;
    } else {
        return -1; // Invalid token type
    }
}

void tokenize(const char *code) {
    char token[MAX_TOKEN_LENGTH];
    int i = 0;
    while (*code != '\0') {
        if (isspace(*code)) {
            code++;
            continue;
        }
        if (is_operator(*code) || is_punctuation(*code)) {
            printf("(%c, OPERATOR)\n", *code);
            code++;
            continue;
        }
        if (isalpha(*code) || *code == '_') {
            while (isalnum(*code) || *code == '_') {
                token[i++] = *code++;
            }
            token[i] = '\0';
            i = 0;
            printf("(%s, %s)\n", token, is_keyword(token) ? "KEYWORD" : "IDENTIFIER");
            continue;
        }
        if (isdigit(*code)) {
            while (isdigit(*code) || *code == '.') {
                token[i++] = *code++;
            }
            token[i] = '\0';
            i = 0;
            printf("(%s, CONSTANT)\n", token);
            continue;
        }
    }
}

```

```

        code++;
    }
}

int main() {
    char code[MAX_CODE_LENGTH];
    printf("Enter your C code:\n");
    fgets(code, MAX_CODE_LENGTH, stdin);
    tokenize(code);
    return 0;
}

```

## OUTPUT:

```

Enter your C code:
printf("hello world")
(printf, IDENTIFIER)
((), OPERATOR)
(hello, IDENTIFIER)
(world, IDENTIFIER)
(), OPERATOR)

```

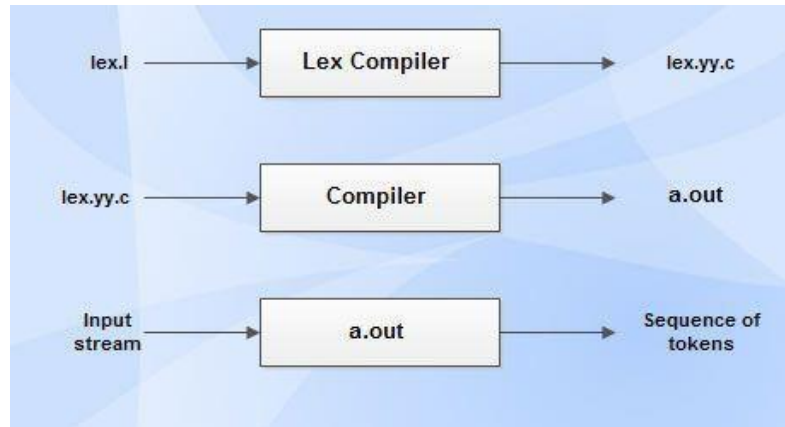
## RESULT:

Thus C program is used implement the program to identify C keywords, identifiers, operators, end statements like; , [ ], { } .

## STUDY OF LEX TOOL

### LEX:

Lex is a tool in lexical analysis phase to recognize tokens using regular expression. Lex tool itself is a lex compiler.



- lex.l is an input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms lex.l to a C program known as lex.yy.c.
- lex.yy.c is compiled by the C compiler to a file called a.out.
- The attribute value can be numeric code, pointer to symbol table or nothing.
- Another tool for lexical analyzer generation is Flex.

### STRUCTURE OF LEX PROGRAMS:

**Declarations:** This section includes declaration of variables, constants and regular definitions.

**Translation rules:** It contains regular expressions and code segments.

Form : Pattern { Action }

Pattern is a regular expression or regular definition.

Action refers to segments of code.

```
%{ LT, LE, EQ, NE, GT, GE, IF,
  THEN, ELSE, ID, NUMBER, RELOP
}%
delim    [ \t\n]
ws       {delim}+
letter   [A-Za-z]
digit    [0-9]
id        {letter}({letter} | {digit})*
number   {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
{ws}     {}
if       { return(IF); }
then     { return(THEN); }
else     { return(ELSE); }
```

**yylex()**

a function implementing the lexical analyzer and returning the token matched

**yytext**

a global pointer variable pointing to the lexeme matched

**yylen**

a global variable giving the length of the lexeme matched

**yyval**

an external global variable storing the attribute of the token

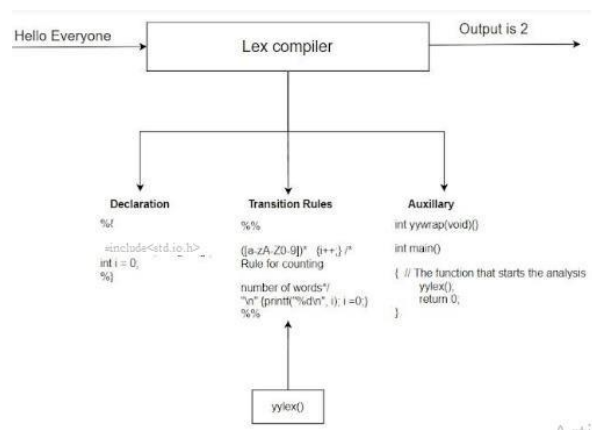
**yywrap:**

Function yywrap is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required.

Declaration-Has header files and initialization of variables

Translation rules-write the rules for counting the words

Auxiliary- has yylex() that calls the translation rules

**PROGRAM:**

```
/*lex program to count number of words*/

% {

#include<stdio.h>

#include<string.h>

int i = 0;

% }

/* Rules Section*/
% %
```

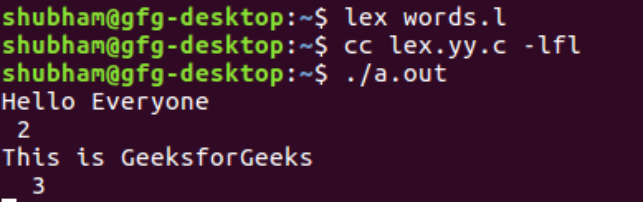
```
([a-zA-Z0-9])* {i++;} /* Rule for counting  
number of words*/
```

```
"\n" {printf("%d\n", i); i = 0;}  
%%
```

```
int yywrap(void){}
```

```
int main()  
{  
    // The function that starts the analysis  
    yylex();  
  
    return 0;  
}
```

### OUTPUT:



```
shubham@gfg-desktop:~$ lex words.l  
shubham@gfg-desktop:~$ cc lex.yy.c -lfl  
shubham@gfg-desktop:~$ ./a.out  
Hello Everyone  
2  
This is GeeksforGeeks  
3
```

### RESULT:

Thus Lex Tool is studied and implemented with sample program.



## EXP 2 - DESIGN A DESK CALCULATOR USING LEX

**AIM:**To Design a Desk Calculator using LEX

### Problem statement:

Create a calculator that performs addition , subtraction, multiplication and division using lex tool.

### ALGORITHM:

- In the headers section declare the variables that is used in the program including header files if necessary.
- In the definitions section assign symbols to the function/computations we use along with REGEX expressions.
- In the rules section assign dig() function to the dig variable declared.
- In the definition section increment the values accordingly to the arithmetic functions respectively.
- In the user defined section convert the string into a number using atof() function.
- Define switch case for different computations.
- Define the main () and yywrap() function.

### PROGRAM:

#### DEFINITION SECTION:

```
%{  
#include <stdio.h>  
#include <stdlib.h>
```

```
int num1, num2;  
char op;
```

```
void dig(char c) {  
    if (isdigit(c))  
        yytext[0] = c;  
}
```

```
%}
```

```
%int yywrap
```

#### RULE SECTION:

```
%%
```

```
[0-9]+    { dig(*yytext); printf("\033[0;32mNumber: %s\033[0m\n", yytext); }  
[-+*/]    { op = *yytext; printf("\033[0;32mOperator: %s\033[0m\n", yytext); }  
\n        { printf("\033[0;34mEnter arithmetic expression (e.g., 2 + 3 * 4):\033[0m\n"); }  
[\t]      { /* ignore whitespace */ }  
          { printf("\033[0;31mInvalid character: %s\033[0m\n", yytext); }  
.
```

```
%%
```

```

int main() {
    printf("\033[0;34mEnter arithmetic expression (e.g., 2 + 3 * 4):\033[0m\n");

    yylex();

    num1 = atoi(yytext);
    num2 = atoi(yytext + strlen(yytext) + 1);

    switch(op) {
        case '+':
            printf("\033[0;34mResult: %d\n\033[0m", num1 + num2);
            break;
        case '-':
            printf("\033[0;34mResult: %d\n\033[0m", num1 - num2);
            break;
        case '*':
            printf("\033[0;34mResult: %d\n\033[0m", num1 * num2);
            break;
        case '/':
            if (num2 != 0)
                printf("\033[0;34mResult: %.2f\n\033[0m", (float)num1 / num2);
            else
                printf("\033[0;31mError: Division by zero\n\033[0m");
            break;
        default:
            printf("\033[0;31mError: Invalid operator\n\033[0m");
            break;
    }

    return 0;
}

int yywrap(void) {
}

```

## OUTPUT:

```

Enter arithmetic expression (e.g., 2 + 3 * 4):
5 + 3 * 2
Number: 5
Operator: +
Number: 3
Operator: *
Number: 2
Result: 11

```

**RESULT:** Thus Design of a Desk Calculator using LEX is implemented

## STUDY OF YACC TOOL

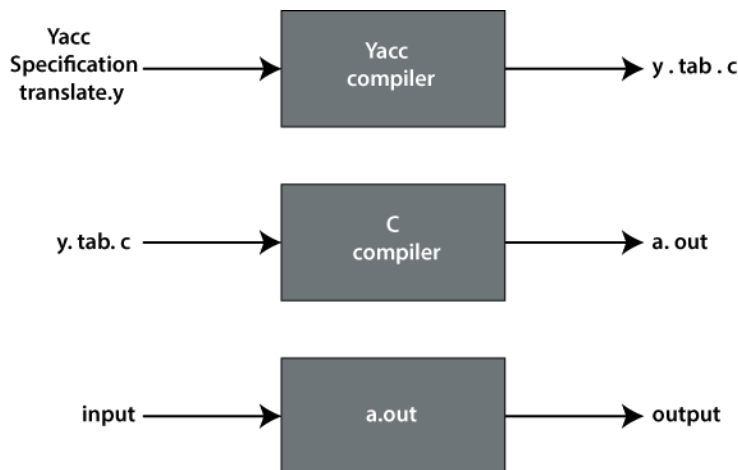
### YACC TOOL:

YACC is known as Yet Another Compiler Compiler. It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar. The input of YACC is the rule or grammar, and the output is a C program. If we have a file translate.y that consists of YACC specification, then the UNIX system command is:

### YACC translate.y

This command converts the file translate.y into a C file y.tab.c. It represents an LALR parser prepared in C with some other user's prepared C routines. By compiling y.tab.c along with the ly library, we will get the desired object program a.out that performs the operation defined by the original YACC program.

The construction of translation using YACC is illustrated in the figure below:



### STRUCTURE OF YACC:

|                   |   |
|-------------------|---|
| C declarations    | <pre>%<br/>#include &lt;stdio.h&gt;<br/>%<br/></pre>  |
| yacc declarations | <pre>%token NAME NUMBER<br/>%<br/></pre>  |
| Grammar rules     | <pre>statement: NAME '=' expression<br/>            expression           { printf("= %d\n", \$1); }<br/>          ;<br/><br/>expression: expression '+' NUMBER { \$\$ = \$1 + \$3; }<br/>            expression '-' NUMBER { \$\$ = \$1 - \$3; }<br/>            NUMBER                { \$\$ = \$1; }<br/>          ;<br/>%<br/></pre> |
| Additional C code | <pre>int yyerror(char *s)<br/>{<br/>    fprintf(stderr, "%s\n", s);<br/>    return 0;<br/>}<br/><br/>int main(void)<br/>{<br/>    yyparse();<br/>    return 0;<br/>}</pre>  |

**Declarations Part:** This part of YACC has two sections; both are optional. The first section has ordinary C declarations, which is delimited by `%{` and `%}`. Any temporary variable used by the second and third sections will be kept in this part. Declaration of grammar tokens also comes in the declaration part. This part defines the tokens that can be used in the later parts of a YACC specification.

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ID NUM
%start expr
```

**Translation Rule Part:** After the first `%%` pair in the YACC specification part, we place the translation rules. Every rule has a grammar production and the associated semantic action. A set of productions:

$$\langle \text{head} \rangle \Rightarrow \langle \text{body} \rangle_1 \mid \langle \text{body} \rangle_2 \mid \dots \mid \langle \text{body} \rangle_n$$

would be written in YACC as

```
<head> : <body>1    {<semantic action>1}
      | <body>2    {<semantic action>2}
      | .....
      | <body>n    {<semantic action>n}
      ;
```

Grammar rule section

```
expr  : expr '+' term
      | term
      ;
term   : term '*' factor
      | factor
      ;
factor : '(' expr ')'
      | ID
      | NUM
```

In a YACC production, an unquoted string of letters and digits that are not considered tokens is treated as non-terminals.

## **EXP 3 - Recognize an arithmetic expression using LEX and YACC**

### **AIM:**

To check whether the arithmetic expression using lex and yacc.

### **ALGORITHM:**

- Using the flex tool, create lex and yacc files.
- In the C include section define the header files required.
- In the rules section define the REGEX expressions along with proper definitions.
- In the user defined section define yywrap() function.
- Declare the yacc file inside it in the C definitions section declare the header files required along with an integer variable valid with value assigned as 1.
- In the Yacc declarations declare the format token num id op.
- In the grammar rules section if the starting string is followed by assigning operator or identifier or number or operator followed by a number or open parenthesis followed by an identifier. The x could be an operator followed by an identifier or operator or no operator then declare that as valid expressions by making the valid stay in 1 itself.
- In the user definition section if the valid is 0 print as Invalid expression in yyerror() and define the main function.

### **LEX AND YACC WORKING**

Parser generator:

- Takes a specification for a context-free grammar.
- Produces code for a parser.

Yacc determines integer representations for tokens:

- Communicated to scanner in file y.tab.h
- use “yacc -d” to produce y.tab.h

Token encodings:

- “end of file” represented by ‘0’;
- A character literal: its ASCII value
- Other tokens: assigned numbers  $\geq 257$ .
- Parser assumes the existence of a function ‘int yylex()’ that implements the scanner.
- Scanner:
  - Return integer value indicates the type of token found
  - Values communicated to the parser using yytext, yylval
  - yytext determines lexeme of a token and yylval determines a integer assigned to a token
  - The token error is reserved for error handling.

Yacc Input File :-

## **int yyparse()**

Called once from main() [user-supplied]

Repeatedly calls yylex() until done:

- On syntax error, calls yyerror() [user-supplied]
- Returns 0 if all of the input was processed
- Returns 1 if aborting due to syntax error.

Example: `int main() { return yyparse(); }`

## **Yacc Grammar Rules :**

### **Rules:**

| Grammar Production         | Yacc Rule               |
|----------------------------|-------------------------|
| <b>A -&gt; B1 B2...Bm</b>  | <b>A: B1 B2 ... Bm;</b> |
| <b>B -&gt; C1 C2,...Cn</b> | <b>B: C1 C2 ... Cn;</b> |
| <b>C -&gt; D1 D2,...Dk</b> | <b>C: D1 D2 ... Dk;</b> |

- Rule RHS can have arbitrary C code embedded within { ... }.

E.g. `A : B1 { printf("after B1\n"); x = 0; } B2 { x++; } B3;`

- Left-recursion more efficient than right-recursion:–

`A : A x | ...` rather than `A : x A | ...`

## **Specifying Operator Properties**

Binary operators: %left, %right, %nonassoc:

Associativity of operator

%left '+' '-'

%left '\*' '/'

%right '^'

Operators in the same group have the same precedence

Unary operators: %prec

- Changes the precedence of a rule to be that of the token specified. E.g.:

%left '+' '-'

%left '\*' '/'

Expr: expr '+' expr

| '-' expr %prec '\*'

## PROGRAM:

```
/* Lex program to recognize valid arithmetic expression
   and identify the identifiers and operators */
%{
#include <stdio.h>
#include <string.h>
    int operators_count = 0, operands_count = 0, valid = 1, top = -1, l = 0, j = 0;
    char operands[10][10], operators[10][10], stack[100];
%}
%%
"(" {
    top++;
    stack[top] = '(';
}
"{" {
    top++;
    stack[top] = '{';
}
"[" {
    top++;
    stack[top] = '[';
}
")" {
    if (stack[top] != '(') {
        valid = 0;
    }
    else if(operands_count>0 && (operands_count-operators_count)!=1){
        valid=0;
    }
    else{
        top--;
        operands_count=1;
        operators_count=0;
    }
}
"}" {
    if (stack[top] != '{') {
        valid = 0;
    }
    else if(operands_count>0 && (operands_count-operators_count)!=1){
        valid=0;
    }
    else{
        top--;
        operands_count=1;
        operators_count=0;
    }
}
"]" {
    if (stack[top] != '[') {
        valid = 0;
    }
    else if(operands_count>0 && (operands_count-operators_count)!=1){
        valid=0;
    }
}
```

```

    }
    else{
        top--;
        operands_count=1;
        operators_count=0;
    }

}
"+"|"-"|"*"|"/" {
    operators_count++;
    strcpy(operators[l], yytext);
    l++;
}
[0-9]+|[a-zA-Z][a-zA-Z0-9_]* {
    operands_count++;
    strcpy(operands[j], yytext);
    j++;
}
%%

int yywrap()
{
    return 1;
}

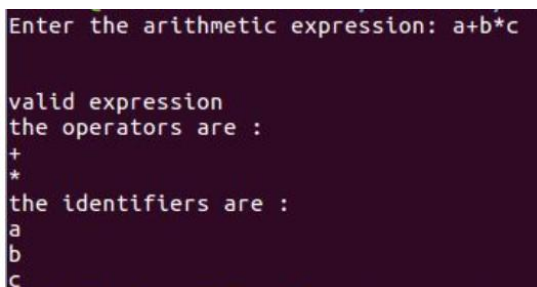
int main()
{
    int k;
    printf("Enter the arithmetic expression: ");
    yylex();

    if (valid == 1 && top == -1) {
        printf("\nValid Expression\n");
    }
    else
        printf("\nInvalid Expression\n");

    return 0;
}

```

### OUTPUT:



```

Enter the arithmetic expression: a+b*c

valid expression
the operators are :
+
*
the identifiers are :
a
b
c

```

**RESULT:** Thus checking whether the arithmetic expression is valid or not using lex and yacc is implemented.



## EXP 4 - EVALUATE EXPRESSION THAT TAKES DIGITS, \*, + USING YACC

### AIM:

To perform arithmetic operations using lex and yacc.

### ALGORITHM:

- Using the flex tool, create lex and yacc files.
- In the definition section of the lex file, declare the required header files along with an external integer variable yyval.
- In the rule section, if the regex pertains to digit convert it into integer and store yyval. Return the number.
- In the user definition section, define the function yywrap()
- In the definition section of the yacc file, declare the required header files along with the flag variables set to zero. Then define a token as number along with left as '+', '-', 'or', '\*', '/', '%' or '(' )'
- In the rules section, create an arithmetic expression as E. Print the result and return zero.
- Define the following:
  - E: E '+' E (add)
  - E: E '-' E (sub)
  - E: E '\*' E (mul)
  - E: E '/' E (div)
  - If it is a single number return the number.
- In driver code, get the input through yyparse(); which is also called as main function.
- Declare yyerror() to handle invalid expressions and exceptions.
- Build lex and yacc files and compile.

### PROGRAM:

#### Lex part:

```
% {  
    /* Definition section*/  
    #include "y.tab.h"  
    extern yyval;  
%}  
  
%%  
[0-9]+ {  
    yyval = atoi(yytext);  
    return NUMBER;  
}  
  
[a-zA-Z]+ { return ID; }  
[ \t]+ ; /*For skipping whitespaces*/  
  
\n { return 0; }  
. { return yytext[0]; }  
  
%%
```

### Yacc part:

```
% {
    /* Definition section */
#include <stdio.h>
% }

%token NUMBER ID
// setting the precedence
// and associativity of operators
%left '+' '-'
%left '*' '/'

/* Rule Section */
%%
E : T      {
                                printf("Result = %d\n", $$);
                                return 0;
                                }

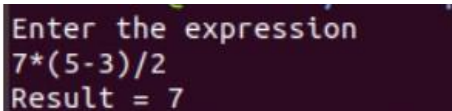
T :
    T '+' T { $$ = $1 + $3; }
  | T '-' T { $$ = $1 - $3; }
  | T '*' T { $$ = $1 * $3; }
  | T '/' T { $$ = $1 / $3; }
  | '-' NUMBER { $$ = -$2; }
  | '-' ID { $$ = -$2; }
  | '(' T ')' { $$ = $2; }
  | NUMBER { $$ = $1; }
  | ID { $$ = $1; };

% %

int main() {
    printf("Enter the expression\n");
    yyparse();
}

/* For printing error messages */
int yyerror(char* s) {
    printf("\nExpression is invalid\n");
}
```

### OUTPUT:



```
Enter the expression
7*(5-3)/2
Result = 7
```

**RESULT:** Thus Evaluation of Arithmetic Expressions using lex and yacc is implemented

## STUDY CODE OPTIMIZATION

**CODE OPTIMIZATION:** The process of code optimization involves

- Eliminating the unwanted code lines
- Rearranging the statements of the code

### CODE OPTIMIZATION TECHNIQUES:

#### 1. Compile Time Evaluation

Two techniques that falls under compile time evaluation are-

##### A) Constant Folding

In this technique,

- As the name suggests, it involves folding the constants.
- The expressions that contain the operands having constant values at compile time are evaluated.
- Those expressions are then replaced with their respective results.

##### B) Constant Propagation

In this technique,

- If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.
- The condition is that the value of variable must not get alter in between.

##### Example:

$\pi = 3.14$

radius = 10

Area of circle =  $\pi \times \text{radius} \times \text{radius}$

Here,

- This technique substitutes the value of variables ' $\pi$ ' and 'radius' at compile time.
- It then evaluates the expression  $3.14 \times 10 \times 10$ .
- The expression is then replaced with its result 314.
- This saves the time at run time.

#### 2. Common Sub-Expression Elimination

The expression that has been already computed before and appears again in the code for computation is called as **Common Sub-Expression**.

In this technique,

- As the name suggests, it involves eliminating the common sub expressions.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

##### Example:

### 3. Code Movement

In this technique,

- As the name suggests, it involves movement of the code.
- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
- Such a code unnecessarily gets execute again and again with each iteration of the loop.
- This leads to the wastage of time at run time.

**Example:**

| Code Before Optimization   | Code After Optimization   |
|--|---|
| <pre>for ( int j = 0 ; j &lt; n ; j ++)<br/>{<br/>x = y + z ;<br/>a[j] = 6 x j<br/>}</pre> | <pre>x = y + z ;<br/>for ( int j = 0 ; j &lt; n ; j ++)<br/>{<br/>a[j] = 6 x j;<br/>}</pre> |

### 4. Dead Code Elimination

In this technique,

- As the name suggests, it involves eliminating the dead code.
- The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

**Example:**

### 5. Strength Reduction

In this technique,

- As the name suggests, it involves reducing the strength of expressions.
- This technique replaces the expensive and costly operators with the simple and cheaper ones.

## **EXP.5 – Generate Three address codes for a given expression (arithmetic expression, flow of control)**

### **AIM:**

To generate the three address code using C program.

### **ALGORITHM:**

- The expression is read from the file using a file pointer
- Each string is read and the total no. of strings in the file is calculated.
- Each string is compared with an operator; if any operator is seen then the previous string and next string are concatenated and stored in a first temporary value and the three address code expression is printed
- Suppose if another operand is seen then the first temporary value is concatenated to the next string using the operator and the expression is printed.
- The final temporary value is replaced to the left operand value.

### **PROGRAM:**

```
#include<stdio.h>
#include<string.h>
void pm();
void plus();
void div();
int i,ch,j,l,addr=100;
char ex[10], exp[10],exp1[10],exp2[10],id1[5],op[5],id2[5];
void main()
{
clrscr();
while(1)
{
printf("\n1.assignment\n2.arithmetic\n3.relational\n4.Exit\nEnter the choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\nEnter the expression with assignment operator:");
scanf("%s",exp);
l=strlen(exp);
exp2[0]='\0';
i=0;
while(exp[i]!='=')
{
i++;
}
strncat(exp2,exp,i);
strrev(exp);
exp1[0]='\0';
strncat(exp1,exp,l-(i+1));
strrev(exp1);
```

```

printf("Three address code:\ntemp=%s\n%s=temp\n",exp1,exp2);
break;

case 2:
printf("\nEnter the expression with arithmetic operator:");
scanf("%s",ex);
strcpy(exp,ex);
l=strlen(exp);
exp1[0]='\0';
break;

case 3:
printf("Enter the expression with relational operator");
scanf("%s%s%s",&id1,&op,&id2);
if(((strcmp(op,"<")==0)||((strcmp(op,">")==0)||((strcmp(op,"<=")==0)||((strcmp(op,">=")==0)||
(strcmp(op,"==")==0)||((strcmp(op,"!=")==0)))))
printf("Expression is error");
else
{
printf("\n%d\tif %s%s%s goto %d",addr,id1,op,id2,addr+3);
addr++;
printf("\n%d\tT:=0",addr);
addr++;
printf("\n%d\tgoto %d",addr,addr+2);
addr++;
printf("\n%d\tT:=1",addr);
}
break;
case 4:
exit(0);
}
}
}
void pm()
{
strrev(exp);
j=l-i-1;
strncat(exp1,exp,j);
strrev(exp1);
printf("Three address code:\ntemp=%s\ntemp1=%c%c%temp\n",exp1,exp[j+1],exp[j]);
}
void div()
{
strncat(exp1,exp,i+2);
printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
}
void plus()
{
strncat(exp1,exp,i+2);
printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3])}

```

## OUTPUT:

```
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:1

Enter the expression with assignment operator:a=b
Three address code:
temp=a
a=temp

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2

Enter the expression with arithmetic operator:a+b
Three address code:
temp=a+b
temp1=temp

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:3
Enter the expression with relational operator:a
>
b

100      if a>b goto 103
101      T:=0
102      goto 104
103      T:=1
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:4
```

## RESULT:

Thus, the three address code was implemented successfully using the C programming language.

## **EXP-6.a – IMPLEMENT CODE OPTIMIZATION TECHNIQUES LIKE DEAD CODE AND COMMON EXPRESSION ELIMINATION**

### **AIM:**

To write a C program to implement the dead code elimination and common expression elimination (code optimization) techniques.

### **ALGORITHM:**

- Start
- Create the input file which contains three address code.
- Open the file in read mode.
- If the file pointer returns NULL, exit the program else go to 5.
- Scan the input symbol from left to right.
- Store the first expression in a string.
- Compare the string with the other expressions in the file.
- If there is a match, remove the expression from the input file.
- Perform these steps 5-8 for all the input symbols in the file.
- Scan the input symbol from the file from left to right.
- Get the operand before the operator from the three address code.
- Check whether the operand is used in any other expression in the three address code.
- If the operand is not used, then eliminate the complete expression from the three address code else go to 14.
- Perform steps 11 to 13 for all the operands in the three address code till end of the file is reached.
- Stop.

### **PROGRAM:**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>

struct op
{
    char l; char
    r[20];
}
op[10], pr[10];

void main()
{
    int a, i, k, j, n, z = 0, m, q;
    char * p, * l;
    char temp, t;
    char * tem;
    clrscr();
    printf("enter no of values");
    scanf("%d", & n);
```



```

for (i = 0; i < n; i++)
{
    printf("\tleft\t"); op[i].l
    = getche();
    printf("\tright\t");
    scanf("%s", op[i].r);
}
printf("intermediate Code\n");for
(i = 0; i < n; i++)
{
    printf("%c=", op[i].l);
    printf("%s\n", op[i].r);
}
for (i = 0; i < n - 1; i++)
{
    pr[z].l = op[n - 1].l;
    printf("\nafter dead code elimination\n");for
    (k = 0; k < z; k++)
    {
        printf("%c\t=", pr[k].l);
        printf("%s\n", pr[k].r);
    }

    for (m = 0; m < z; m++)
    {
        tem = pr[m].r;
        for (j = m + 1; j < z; j++)
        {
            p = strstr(tem, pr[j].r);if
            (p)
            {
                t = pr[j].l; pr[j].l
                = pr[m].l;
                for (i = 0; i < z; i++)
                {
                    l = strchr(pr[i].r, t);if
                    (l) {
                        a = l - pr[i].r;
                        //printf("pos: %d", a);
                        pr[i].r[a] = pr[m].l;
                    }
                }
            }
        }
        printf("eliminate common expression\n");for
        (i = 0; i < z; i++) {
            printf("%c\t=", pr[i].l);
            printf("%s\n", pr[i].r);
        }

        for (i = 0; i < z; i++)
        {
            for (j = i + 1; j < z; j++)

```

```

{
    q = strcmp(pr[i].r, pr[j].r);
    if ((pr[i].l == pr[j].l) && !q)

        {
            pr[i].l = '\0';
            strcpy(pr[i].r, '\0');
        }
    }
}
printf("optimized code");
for (i = 0; i < z; i++)
{
    if (pr[i].l != '\0') {
        printf("%c=", pr[i].l);
        printf("%s\n", pr[i].r);
    } } getch();
}

```

## OUTPUT:

```

Enter the number of values: 5

Left:
Right: 5

Left:
Right: j

Left:
Right: ^C
[cduser42@localhost ~]$ ./a.out
Enter the number of values: 3

Left:
Right: a

Left:
Right: xyz

Left:
Right: def

Intermediate Code:

= a

= xyz

= def

After Dead Code Elimination:

= def

Eliminate Common Expression:

= def

Optimized Code:

= def

```

**RESULT:** Thus the above program was compiled and executed successfully and output is verified.

## EXP.6.b – IMPLEMENT CODE OPTIMIZATION TECHNIQUES – COPY PROPAGATION

### AIM:

To write a C program to implement copy propagation.

### ALGORITHM:

- The desired header files are declared.
- The two file pointers are initialized one for reading the C program from the file and one for writing the converted program with constant folding.
- The file is read and checked if there are any digits or operands present.
- If there is, then the evaluations are to be computed in switch case and stored.
- Copy the stored data to another file.
- Print the copied data file.

### PROGRAM:

```
/*CODE OPTIMIZATION - CONSTANT FOLDING*/
#include<stdio.h>
#include<string.h>
void main() {
    char s[20];
    char flag[20]="//Constant";
    char result,equal,operator;
    double op1,op2,interrslt;
    int a,flag2=0;
    FILE *fp1,*fp2;

    fp1 = fopen("input.txt","r");
    fp2 = fopen("output.txt","w");

    fscanf(fp1,"%s",s);
    while(!feof(fp1)) {
        if(strcmp(s,flag)==0) {
            flag2 = 1;
        }
        if(flag2==1) {
            fscanf(fp1,"%s",s);
            result=s[0];
            equal=s[1];
            if(isdigit(s[2])&& isdigit(s[4])) {
                if(s[3]=='+'||s[3]=='-'||s[3]=='*'||s[3]=='/') {
                    operator=s[3];
                    switch(operator) {
                        case '+':
                            interrslt=(s[2]-48)+(s[4]-48);
                            break;
                        case '-':
                            interrslt=(s[2]-48)-(s[4]-48);
                            break;
                        case '*':
```

```

                                interrslt=(s[2]-48)*(s[4]-48);
                                break;
                        case '/':
                                interrslt=(s[2]-48)/(s[4]-48);
                                break;
                        default:
                                interrslt = 0;
                                break;
                }
                fprintf(fp2,"/*Constant Folding*\n");
                fprintf(fp2,"%c = %lf\n",result,interrslt);
                flag2 = 0;
        }
    } else {
        fprintf(fp2,"Not Optimized\n");
        fprintf(fp2,"%s\n",s);
    }
} else {
    fprintf(fp2,"%s\n",s);
}
fscanf(fp1,"%s",s);
}
fclose(fp1);
fclose(fp2);
}

```

### OUTPUT:

```

[student@localhost ~]$ gcc codeopt.c
[student@localhost ~]$ ./a.out [student@localhost ~]
$cat input.txt

#include<stdio.h>
int main()
{
//Constant a=2+4; b=a+10;
}
[student@localhost ~]
$cat Output.txt

#include<stdio.h> int main()
{
/*Constant Folding*/ a = 6.000000 b=a+10;
}

```

### RESULT:

Thus the C program was written for copy propagation – A Code Optimization Technique.

## **EXP.7 – Generate Target Code (Assembly language) for the given set of Three Address Code**

### **AIM:**

To generate assembly language for the three address code using C program.

### **ALGORITHM:**

- Get address code sequence.
- Determine current location of 3 using address (for 1st operand).
- If the current location does not already exist, generate move (B, O).
- Update address of A (for 2nd operand).
- If the current value of B and () is null, exist.
- If they generate operator () A, 3 ADPR.
- Store the move instruction in memory.

### **PROGRAM:**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Function to generate assembly code from three-address code
void generateAssembly(char three_address_code[][50], int num_instructions) {
    // Iterate through each instruction
    for (int i = 0; i < num_instructions; i++) {
        char op[10], arg1[20], arg2[20], result[20];
        // Parse the instruction
        sscanf(three_address_code[i], "%s %s %s %s", op, arg1, arg2, result);

        // Generate assembly code based on the operation
        if (strcmp(op, "ADD") == 0) {
            printf("MOV AX, %s\n", arg1);
            printf("ADD AX, %s\n", arg2);
            printf("MOV %s, AX\n", result);
        } else if (strcmp(op, "SUB") == 0) {
            printf("MOV AX, %s\n", arg1);
            printf("SUB AX, %s\n", arg2);
            printf("MOV %s, AX\n", result);
        } else if (strcmp(op, "MUL") == 0) {
            printf("MOV AX, %s\n", arg1);
            printf("IMUL %s\n", arg2);
            printf("MOV %s, AX\n", result);
        } else if (strcmp(op, "DIV") == 0) {
            printf("MOV AX, %s\n", arg1);
            printf("IDIV %s\n", arg2);
            printf("MOV %s, AX\n", result);
        } else {
            printf("Unsupported operation: %s\n", op);
        }
    }
}

int main() {
    char three_address_code[100][50]; // Assuming max 100 instructions and 50 characters per instruction
```

```

int num_instructions = 0;

// Get three-address code instructions from the user
printf("Enter three-address code instructions (type 'END' to finish):\n");
char line[50];
while (1) {
    fgets(line, sizeof(line), stdin);
    if (strcmp(line, "END\n") == 0) {
        break;
    }
    strcpy(three_address_code[num_instructions++], line);
}

// Generate assembly language instructions
generateAssembly(three_address_code, num_instructions);

return 0;
}

```

## OUTPUT:

```

Enter three-address code instructions (type 'END' to finish):
ADD A B C
SUB D A E
MUL F G H
END
ADD A B C
SUB D A E
MUL F G H
END

MOV AX, A
ADD AX, B
MOV C, AX
MOV AX, A
ADD AX, B
MOV C, AX
MOV AX, D
SUB AX, A
MOV E, AX
MOV AX, D
SUB AX, A
MOV E, AX
MOV AX, F
IMUL G
MOV H, AX
MOV AX, F
IMUL G
MOV H, AX

```

**RESULT:** Thus generation assembly language for the three address code using C program is implemented.







